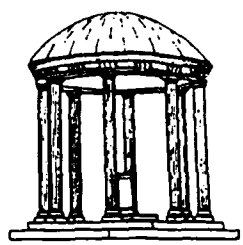
The COOL Library: User's Manual, Version 1.0

CO02

James Coggins

SoftLab Software Systems Laboratory
The University of North Carolina
Department of Computer Science
CB#3175, Sitterson Hall
Chapel Hill, NC 27599-3175



Supported in part by the NASA Center of Excellence in Space Data and Information Sciences, contract#500-62 and the Office of Naval Research, contract#N-00014-86-K-0680.

Reference and title

CO02

The COOL Library: User's Manual. Version 1.0

Publication history

Version 1.0: October 1989

Copyright and trademarks

Authors

James Coggins

The COOL Library
User's Manual

Table of Contents

Acknowledgements

The COOL library and the documentation for it has been one of my principal research activities for the three years since I arrived at the University of North Carolina. When I moved to Chapel Hill, I lost overnight the entire research software library I had been developing and using due to incompatibilities between the language and computing environments I had worked in for eight years and the UNIX environment at UNC. Many graduate students and faculty colleagues helped me to learn C++ and to begin building the library and its documentation. I am pleased to acknowledge their contributions here:

Tim Rentsch introduced me to the "lunatic fringe" of Object-Oriented Design and helped with some of the initial, critical decisions underlying some of the best innovations in the library. Jonathan Walker pointed the way toward complex but innovative and effective directory and makefile organizations and extolled some of the virtues of C++ to me when I was but a novice. Brice Tebbs explored and taught me basics of C++. Murugappan Palaniappan implemented and reimplemented many of the early classes as my design concepts refined through experience. His work appears in many places in the library, often in places other than where he wrote them. John Rohlf wrote the first version of the graphics classes. Andrew Glassner helped me to debug some of the graphics procedures by teaching part of our intro graphics course using a rapidly evolving COOL. The patience of the students in that class also deserves mention as COOL went through a rapid evolution (revolution?) between their assignments using the library. Greg Bollella's work through our department's Softlab to help me set up the administrative structures of COOL resulted in publication of our approach both on the Internet and in SIGPLAN Notices; this paper is substantially reproduced in Chapter 4 of the Supplementary Documents. Greg's creative expertise with Makefiles will be further illustrated in a later public release of COOL in which the Makefile structure will support multiple architectures. Cathy Vishnevsky and James Chung did their required Writing Projects on C++ and COOL, respectively, and some portions of the chapter in teh Supplementary Documents on C++ have been influenced by their reports.

My local users group, a captive audience whose sweat and time have been spent exploring, using, and sometimes working around COOL have earned a mention here, too: K. C. Low, Eric Fredericksen, Lisa Baxter, Yuchin Fu, Amandeep Jawa have been and are yet exploring the use of the library in our research. Help, comments and suggestions of various kinds have been received from Jonathan Leech, Graham Gash, Tim Cullip, John Gauch, Brian White, Russell Taylor, Michael Kelley, Steve Pizer, and Fred Brooks.

When I interviewed for a faculty position at UNC, Fred Brooks asked me how my computer vision research might shed any light on the problems of software engineering. I didn't have a very good answer then. Fred, I would like hereby to amend that response.

The COOL Library: User's Manual

Introduction

This report describes each of the sublibraries and classes of COOL, a library written in the C++ Programming Language supporting image pattern recognition and computer graphics research. This report documents the structures of the sublibraries of COOL, including their component classes, design decisions reflected in the current implementation, and prospective improvements to the sublibraries.

The design principles and criteria used in COOL, an introduction to C++, and practical library management techniques, as well as examples of programs using COOL are found in the companion report, The COOL Library: Supplementary Documents.

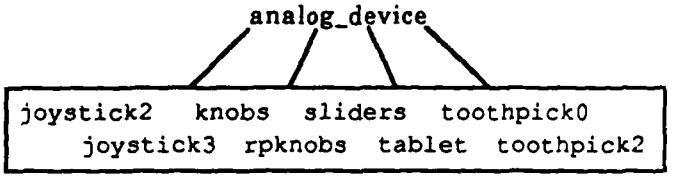
Each class' objective will be stated briefly, then the constructors and member functions will be described. A "Commentary" section will discuss, for most classes, possible or anticipated improvements, design decisions that seem particularly good or poor, and usage notes.

Clearly, this report will be revised frequently as experience with COOL accumulates. Therefore, the printing date is used in the header of each page. Also, the sublibrary name is written in the header line for easier indexing. The sublibraries are alphabetically ordered. Classes within each sublibrary are listed in order of a depth-first inheritance tree traversal, where appropriate, and alphabetically otherwise.

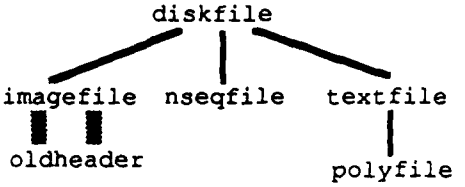
The next page is a diagram showing the inheritance structure of COOL and its organization into sublibraries. The portions of the diagram illustrating each sublibrary will be used to introduce the sublibrary.

The Structure of COOL

Analoglib



Disklib



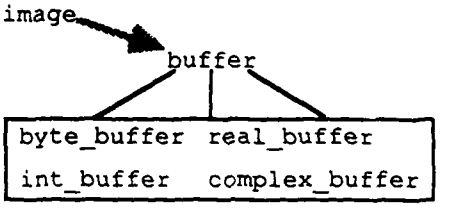
Displaylib

ikonas imagetool
Ximagetool

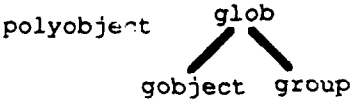
Enzymelib

fft_server

Imagelib/Bufferlib

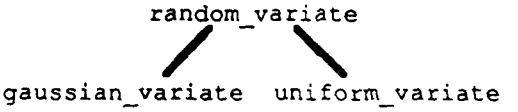


Graphlib

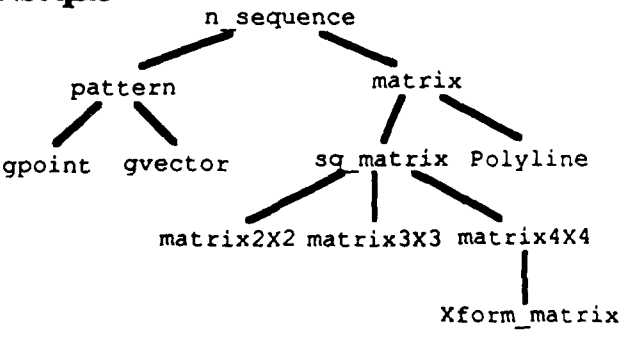


Misclib

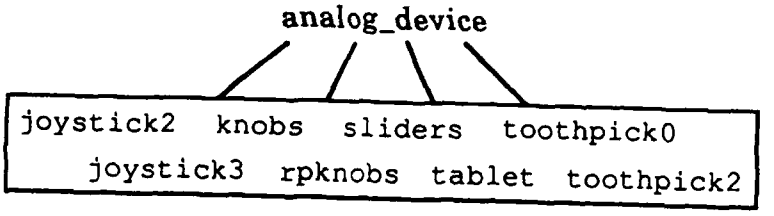
complex
magpha
histogram
nd_gaussian
subscript
timer



Nseqlib



Analoglib



Analoglib contains classes that encapsulate a variety of human/computer interaction devices in the Graphics and Image Laboratory in the Computer Science Department at UNC-Chapel Hill.

Note: The code of Analoglib is distributed with COOL because it illustrates an interesting style of encapsulation, but since these classes are quite specific to the UNC environment, the Makefiles in the COOL distribution do not compile these classes or archive them in the `cool.a` library archive.

Class analog_device

analog_device is an abstract superclass that provides services to the device-specific subclasses.

Constructors

The protected constructor,
analog_device(int type, int nb, char* devicedriver)
is invoked by the constructors of the subclasses to initialize the requested device and to declare the communication protocol to be used with the device. The parameter type indicates the storage class of the data values returned from the device where 0 means char, 1 means short, and 2 means int. The parameter nb indicates the number of bytes expected in a single read of the device, and the devicedriver parameter is a null-terminated character string containing the name of the device driver (as in "/dev/ad2d0" for the 2D joystick).

By protecting the constructor, we ensure that the user cannot create an analog_device object directly, which would be a serious error.

The destructor ~analog_device() simply closes the device driver.

Member functions

analog_device& poll()
: reads the device and reports an error to cerr if the number of bytes received is different from the number of bytes expected as specified by the device-specific constructor.

rawvalue(int i)
: returns the ith value in the buffer, interpreting the array of bytes as char, short, or int as was indicated by the device constructor. The reinterpretation of the array of bytes as different types is performed using a union ad_buffertype defined in analog_device.h. An invocation of rawvalue does **not** poll the device.

Device-specific classes: `public analog_device`

`joystick2`	2-dimensional joystick
`joystick3`	3-dimensional joystick
`knobs`	a series of 4 knobs on the joystick box
`rpknobs`	a group of 8 knobs on the toothpick box
`sliders`	a set of 4 sliders on the joystick box
`toothpick0`	the toothpick on the gray box with the rpknobs
`toothpick2`	the black toothpick
`tablet`	the data tablet and the buttons on its puck

The structure of the device-specific classes are sufficiently similar that they will be discussed together. The header files of each device class contain symbol definitions for use with the class. For example, `knobs.h` defines symbols `KNOB0` through `KNOB3`. Some definitions, such as the colors for the sliders, `RED`, `GREEN`, `BLUE`, and `WHITE`, are defined in file `cool.h` because they are shared by several parts of COOL.

Constructors

Each class contains a null constructor and a destructor. The action of the constructor is to pass the appropriate arguments to the `analog_device` constructor. These are hard-coded, so no user action is required.

Member functions

Values may be obtained from the device in three ways. The function `value(int i)` polls the device and returns the ith value received. `operator>>(int i)` polls the device and returns the ith value received.

Some devices also have a series of messages that obtain a specific value by name from the device. These are listed below for each device.

```
knobs     knob0()    knob1()    knob2()    knob3().
rpknobs   rpknob0()  rpknob1()  rpknob2()  rpknob3()
          rpknob4()  rpknob5()  rpknob6()  rpknob7()
sliders   red()      green()    blue()     white()
tablet    yellow()   white()    blue()     green()    z()
```

The values returned by all of the member functions are normalized. `Knobs`, `rpknobs`, and `sliders` return scalars in the range 0.0 to 1.0. The 2D joystick can return a single position coordinate or both in the form of a `gpoint` object in the unit square with corners (0,0) and (1,1). The 3D joystick can return single coordinates or a `gvector` object in (0,0) to (1,1). The types of values returned by the joystick classes are based on characteristics of these joysticks. The 2D joystick is a positional device, so its return value is a point; the 3D joystick is a spring-loaded velocity device, so its value is returned as a vector. The `tablet` can return the position as a `gpoint` and the button values as `int`s. I have never completely

understood how the toothpicks work anyway, so I just copied the conventions I saw in some programs in the lab.

Commentary

The inheritance structure appears to be a good idea. The superclass provides just the services needed by the device classes, and there is some meaningful code sharing. The ability to specify parameters to the superclass constructor from the derived class constructor in the constructor's procedure header made this class hierarchy work.

This class hierarchy has not been used much in practice because the analog devices are attached to only one particular computer system, and that system is much less used because workstations and window widgets have become a more common working environment. Nevertheless, applications arise occaisionally where the best way to proceed is indeed to use a physical device - taping down a paper onto the tablet for tracing, for example - and for these purposes, this sublibrary is just what is needed to get the application written while hiding messy details like bytes and device driver names.

It is not clear, because of the lack of experience with this library, that the decision to provide both symbol definitions and named messages for each value in some classes was a good one.

Displaylib

ikonas imagetool Ximagetool

Displaylib contains three classes that operate graphical display systems in our laboratory. The Adage/Ikonas 3600 display system is operated by class `ikonas`. Windows on color SUN workstations running the SunView environment are operated by class `imagetool`. Windows on Suns or DEC 3100 workstations running th X Windows System are operated by `Ximagetool`.

Note: The classes in `displaylib` are device-specific and require support libraries to operate. These classes are distributed with COOL, but only the `imagetool` and `Ximagetool` classes are compiled and included in the `cool.a` file.

Classes `ikonas`, `imagetool`, and `Ximagetool`

Since there are many similarities among these three classes, they will be discussed together and significant additions or deviations will be noted as required.

Constructors

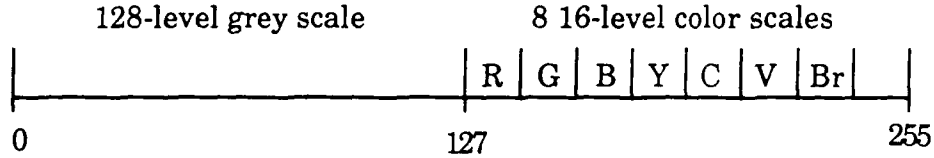
The constructors of all three classes initialize their devices, including their color lookup tables. The screen area initialized by all three classes is 512x512 pixels. The `imagetool` and `Ximagetool` classes have an additional constructor that allows the user to specify the row and column coordinates of the upper left corner of this 512x512 window.

The display window is considered to be composed of four 256x256 viewports numbered rowwise as follows:

```
0 1
2 3.
```

Thus, even-numbered viewports are on the left and odd-numbered viewports are on the right. These viewport numbers are used in the member functions described below.

The color table is initialized by the constructor to a configuration, unique to our lab as far as we know, that we have found particularly useful in our research. Both devices provide an 8-bit lookup table that maps the integers 0 to 255 into 24-bit colors (8 bits each to drive the red, green, and blue guns). We allocate these 256 driving levels as indicated below.



A 128-level gray scale occupies driving levels 0 to 127. The rest of the range is divided into eight 16-level color scales. Each color scale contains a particular hue at full saturation that moves through 16 levels of brightness. The colors in our implementation are red, green, blue, yellow, cyan, violet, and brown. The last color scale is a rather attractive cornflower that is reserved for system use.

The destructors close the devices. On the Ikonas, the image remains on the screen after closing; the `imagetool` and `Ximagetool` destructors remove the display window entirely.

Member functions

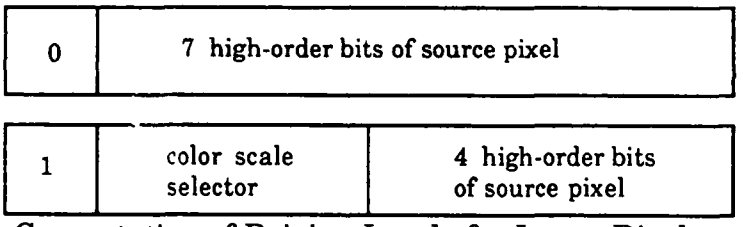
The member functions for these classes fall into three groups: clear, image display, and graphical display.

clear(int viewport, int color)
sets the specified viewport (default to viewport 0) to a particular color value, defaulted to 0 (black).

clear_all(int color)
sets all four viewports to the indicated color, defaulted to 0 (black).

display(int vport, image src)
displays the source image, which may have any storage type, in the indicated viewport. No intensity scaling is performed by `display`, but a type conversion to `byte` is performed. The byte image is mapped into the gray scale part of the color table only.

display(int vport, image src, image mask, byte enable)
displays image `src` in the indicated viewport and uses the `mask` image and the `enable` byte to determine what color scale is used at each pixel. Both images are converted to `BYTE`. Then the driving value is computed for each pixel. After ANDing the `mask` pixel with the `enable` byte, the lowest-order 1 bit in the byte determines which color scale is used for that pixel. The lowest order ('1') bit maps to red, the '2' bit to green, and so on. If after ANDing the byte is zero, the grey scale is used. The particular value to use within the chosen scale is determined from the high-order bits of the source pixel (7 bits if the gray scale is used, 4 bits if a color scale is used.



Computation of Driving Levels for Image Display
top: grey scale mode
bottom: color scale mode

void display0(int vport, image im)
is a direct, uninterpreted byte dump to the screen. `Display0` is invoked by the other display functions and could probably be declared as a private function, but I felt it was too potentially useful to hide.

The last group of functions display graphical objects, currently as wire-frames.

display(int vport, polyobject* obj, int color)
displays a `polyobject`, which is a collection of `Polylines`.

display(int vport, Polyline* poly, int color)
displays a single `Polyline` structure, which is a sequence of connected line segments. The `color` parameter defaults to white in both cases. The `Polyline` class is capitalized because of a name conflict with the SunTools

procedure `polyline`. These display procedures assume that all necessary projections have been performed. The graphical objects should be scaled to the unit square with corners (0,0) and (1,1) to match the size of the viewport. If the graphical object exceeds this size, it will be drawn over the other viewports. Thus, to produce graphical output that covers all four viewports, one can scale the graph to the square with corners (0,0) and (2,2) and display it in viewport 2.

The `imagetool` and `Ximagetool` classes (and not the `ikonas` class) has one more function for handling interactive input.

`int get_input(int& button, int& vport, subscript& point)`
checks for mouse clicks in the imagetool window and returns 0 if there have been none, 1 if the mouse has been clicked in the window. In addition, it places in its arguments the number of the button that was pressed (0 means left, 1 means center, 2 means right), the viewport number, and the (row,col) subscript indicating the location within the viewport where the button was pressed. This function can be used to control a loop waiting for, say, the right button to be pressed in window 2. While `get_input` is running in such a loop, the window can be moved, redisplayed, or even resized. In such a loop, other mouse clicks may be used to control the program in other ways, including printing pixel values or storing locations as vertices of a contour.

Commentary

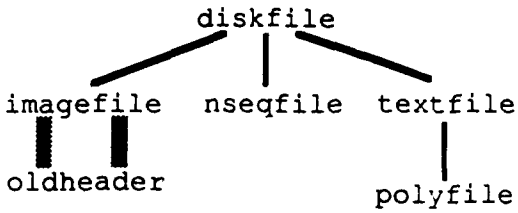
When I first wrote `displaylib`, before the COOL library management strategy was developed, I included an abstract superclass, `display_device`, that encapsulated the many commonalities between the `ikonas` and `imagetool` classes. I found that the entire support libraries for the Ikonas and for SunView were being linked into my programs. This was clearly unacceptable, so I removed `display_device` and left the device classes separate. It may be time to try the abstract superclass again.

When COOL was first written, SunWindows was the dominant window management system at UNC. Since then, the X Window System has become the dominant window system. The development of `Ximagetool` (through modification of `imagetool`) required only a day and would have been much quicker except for compatibility problems with function prototypes in the header files for X.

These classes have served my needs well. Others may require control of the size of the `imagetool` window or control over special features of other display systems. The lack, at present, of facilities in COOL for shaded surface graphics is a serious limitation that would require changes here and in `graphlib`. I would particularly like to incorporate the Pixar Image Computer as one of the supported display devices, especially since we have developed network access methods to the Pixar so it can be run from workstations other than its host computer.

`Displaylib` illustrates my approach to the objective of separation of concerns. Since display and interaction involve concepts and concerns not shared with data structures for processing or for storage and communication, I seek to separate activities in each of those three categories. The good side of this decision is that the concerns are separated and what is learned about display for one class can be shared in writing display code for other classes. For example, having written display code for images made writing the display code for graphics easy. In addition, the different display procedures are able to share services, especially the initialization services provided by the constructors, and the input services in `imagetool`. (For an even better example, see `disklib`.) Thus, an `image` or a `polyobject` does not know how to display itself; instead, these computational structures are dispatched to a display device that knows how to map the object onto a display. It might appear that knowledge of the internal structure of displayable classes must be contained in the display device class, but this is not really so. The display class needs simply to know how to ask for what it needs using data access member functions. With that design, changes in data formats, styles or storage patterns does not affect the interface used by the display device.

Disklib



Disklib encapsulates UNIX disk file handling procedures for various kinds of files needed in COOL. `Diskfile` provides basic I/O services while the subclasses encapsulate various file formatting conventions.

Class diskfile

Diskfile is an abstract superclass that provides for its subclasses the basic disk services: open, close, read, write, and seek.

Constructors

Three constructors are provided: the null constructor, diskfile(char*) to open the named file for reading, and diskfile(char*,int) to open the named file for the type of access indicated by the integer permission code,

The null constructor does not open a file and leaves a flag to that effect. The file may be opened later with the diskopen(char*,int) message. The diskfile(char*) constructor opens the named file for READ access. The diskfile(char*,int) constructor opens the file named in the character string with the permission codes given in the integer argument.

Symbols READ, CREATE, and UPDATE are defined for the integer permission codes.

The destructor simply closes the file.

Member functions

void filename(char*)
: copies the file name to the argument. It is assumed that an appropriate buffer is allocated.

void diskopen(char*,int)
: is used to open a file with the integer permissions specified when the diskfile was created with the null constructor.

int diskread(char* buf, int len)
: attempts to read len bytes from the file into the buffer buf, returning the actual number of bytes read.

int diskwrite(char* buf, int len)
: attempts to write len bytes to the file from the buffer buf, returning the actual number of bytes written.

int diskseek(long pos,int whence)
: performs a seek to position pos relative to the beginning, end, or current position in the file depending on the value of whence. Symbols FROMBEG, FROMEND, and FROMHERE are defined for use in this argument.

Commentary

I originally wrote the image object with load() and store() member functions that would read images from disk and store them there. These were the longest member functions in the whole image class, so they already looked out of place there. When I later found that I needed to load() and store() certain matrices,

I realized that my experience in implementing reads and writes of `images` could not be used to help me define analogous operations on matrices. This struck me immediately as a serious design flaw. I decided to bring together all disk handling routines under a single inheritance structure in which basic disk file services are provided by a base class while file formatting information for different kinds of files is hidden in the subclasses. This design decision has been a big win for COOL. This experience led to the formulation of the separation of concerns that serves as a principal design criterion for COOL: the separation between processing, storage and communication, and display and interaction. This idea is applied in several sublibraries of COOL.

An enhancement to this library that is in the works is the use of the XDR protocols to read and write disk files. XDR translates byte sequences between different architectures so that files can be read by processes on any supported architecture independent of the architecture of the machine that created the file. This is an important capability in our lab where multiple architectures are routinely used.

Classes imagefile:public diskfile and oldheader

Imagefile and oldheader jointly implement the storage of 2D and 3D images.

Constructors

imagefile()
: prompts the user for a file name to be opened with READ access.

imagefile(char*)
: opens the named file for read access.

imagefile(char*,int)
: opens the named file for the access indicated.

imagefile(buffer_type, subscript,int)
: prompts the user for a file name to be opened (created) for writing with the stated buffer_type, 2D size and number of planes.

imagefile(char*,buffer_type, subscript,int)
: opens (creates) the named file for writing with the stated buffer_type, 2D size and number of planes.

Symbols READ, CREATE, and UPDATE are defined for the integer permission codes.

buffer_type is a typedef equivalent to char*. A buffer_type is a one-character-long string indicating a storage type. Symbols are defined as follows for the implemented values of buffer_type: BYTE, INTEGER, REAL, COMPLEX, GREYTYPE (same as INTEGER).

Member functions

buffer_type type()
: returns the type of the image(s) contained in the imagefile.

void size(subscript& s, int& p)
: returns in its arguments the 2D shape of the image planes and the number of planes in the imagefile.

image load(int plane=0)
: returns the indicated plane from the imagefile. The image returned has the storage type indicated by type() and the size indicated by size().

save(image im, int plane)
: saves the given image in the indicated plane of the imagefile.

Commentary

Class oldheader encapsulates the strange header format used in the UNC-developed /usr/image library. Its operation is invisible to the user, and in fact, no support is provided in COOL for any of the header functions other than what is described above. Oldheader is a separate class in order to keep header-format-dependent junk out of the imagefile code, which is nicely compact and elegant.

We expect to change the header format soon, so I have encapsulated it so I know what has to be changed.

The linkage between an `imagefile` and its `oldheader` is of some technical interst. The `imagefile` contains a pointer to its `oldheader`, and when the `oldheader` is constructed, it is provided a pointer back to the `imagefile`. Now since `oldheader` and `imagefile` are **friend** classes, each can invoke any methods or access any data of the other. This intimate relationship keeps the operations of the classes cleaner and is justified since their separation is purely formal; they are really parts of the same conceptual object.

The functionality supported now is the minimum that is required. An `imagefile` stores a series of 2D planes of pixels; all of the planes are of the same storage type and same size. The `imagefile` may be interpreted as a series of 2D `images` or as a single 3D `image`, as desired, but this interpretation must be handled in the client software - dimensionality above 3 is not handled in the `imagefile` class at all for now. It is possible to make `imagefile` handle higher-dimensional `images` and the new header format might provide such functionality.

`Imagefile` reads only whole planes, but there is some demand for a capability to read portions of `image` planes. This feature may be added in the future.

The current `imagefile` class has proven very easy to use and extremely valuable. Many nettlesome details are hidden away in this class, making it one of the most satisfying encapsulations in the library.

Class nseqfile:public diskfile

Nseqfile allows the storage to disk of data contained in a class that is part of the Linear Algebra Tree (sublibrary nseqlib).

Constructors

nseqfile(char*,int)
: opens the named file for the access indicated by the integer permission code.

Symbols READ, CREATE, and UPDATE are defined for the integer permission codes.

Member functions

load(n_sequence&)
: loads the file into the given n_sequence. The argument could be an object of any subclass of n_sequence.

save(n_sequence&)
: saves the given n_sequence to disk. The argument could be an object of any subclass of n_sequence.

Commentary

The use in the member functions of arguments rather than function returns is dictated by the C++ type system. It is essential that any class in the Linear Algebra Tree be valid for use with the nseqfile, and this was the way to achieve that end.

Class `textfile:public diskfile`

This class adds to `diskfile` operations that are valid for text files.

Constructors

`textfile(char*,int)`
: Opens the named file for the access indicated by the integer permission code.

Symbols `READ`, `CREATE`, and `UPDATE` are defined for the integer permission codes.

Member functions

`int read_int()`
: Reads an integer from the file

`double read_real()`
: Reads a double from the file

`void real_eol()`
: Reads to the end of the current line immediately

`void write(int)`
: Writes an integer to the current line

`void write(double)`
: Writes a double to the current line

`void newline()`
: Writes a newline to the file

Commentary

The `textfile` class provides operations that interpret the bytes as strings representing numbers. The `newline()` and `read_eol()` functions are useful for formatting. Spaces are written after the number in both `write()` functions.

Class `polyfile:public textfile`

A polyfile is a textfile that understands how polyobjects are stored on disk and thus knows how to load and store them.

Constructors

`polyfile(char*,int)`
: Opens the named file for the access indicated by the integer permission code.

Symbols `READ`, `CREATE`, and `UPDATE` are defined for the integer permission codes.

Member functions

`polyobject* load()`
: Loads a polyobject from disk.

`save(polyobject*)`
: Saves a polyobject to disk.

Commentary

This simple class and its interactions with its subclasses is a marvelous example of encapsulation and inheritance. All that the application needs is to load and save polyobjects, and that is exactly what is provided, with a minimum of excess administrative code.

A polyobject is stored as follows:
integer number of polylines in the polyobject
  integer number of points in the first polyline
    <the points as triples of reals>
  integer number of points in the second polyline
    <the points as triples of reals>
  ...
Anything after the last required number on a line is ignored, so comments can be placed at the end of each line.

Enzymelib

`fft_server`

An enzyme is an encapsulation of a well-defined procedure. This characterization applies in biochemistry as well as it does to object-oriented design. Enzymelib contains process encapsulations: classes whose purpose is not in the data it holds but in the effects it has on objects of other classes. My enzymelib contains several process encapsulations of interest to my research, but perhaps only temporary interest even there. The class we are distributing with COOL is one of more general interest and one that epitomizes the value of process encapsulation: the `fft_server` class.

Class `fft_server`

The `fft_server` class computes 2D discrete, fast discrete Fourier transforms and inverse transforms on objects of class `image`.

Constructors

The fft_server constructor takes two integer arguments indicating the row and column dimensions of the image this server will transform. The dimensions must be powers of 2 but not necessarily equal. Given this information, the constructor precomputes several tables of indexes and complex coefficients that will be used to speed the computation when the `fft_server` is called upon to operate. The `fft_server` can operate only on images of the size specified in the constructor. If you need FDFTs of different sizes in the same program, you must create separate `fft_server` objects for each size.

The destructor simply deletes the tables created by the constructor.

Member functions

Two data access functions are provided.

`int dim()`
: returns the dimensionality of this `fft_server`, 1 or 2 (always 2 right now)

`int size()`
: returns the product of the number of rows and the number of columns. These particularly unhelpful functions remain in the code out of interia.

`image forward(image im)`
`image inverse(image im)`
: These functions do the work of the FDFT and inverse FDFT respectively. The input image may be of any storage type; it will be converted to storage type `COMPLEX` internally. The input image is not modified by the FFT computation. Both functions yield an image that is in `COMPLEX` storage format.

Private member functions

The `fft_server` contains several private member functions. `server_error` reports errors to `ostream cout`.

`make_server(int,int)`
: is invoked by the constructors to create the tables of subscripts and complex coefficients.

`fft(image)`
: controls the computation of a forward FDFT by calling the other private functions.

`fft_shuffle(image)`
: performs an in-place bit-reverse shuffle of the argument image.

`fft_atno(image)`

performs a fast discrete Fourier array transform using an algorithm optimized for 2D images published by L. Johnson and A. Jain in IEEE PAMI.

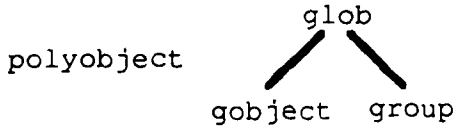
Commentary

The `fft_server` class works very well on 2D images of size up to 1/4 million pixels (e.g. 512x512 images). At that size, the computation slows to unreasonable times. The `fft_server` class could be extended trivially to handle 1-D transforms of `patterns`. This need simply has not arisen yet in our work.

The notion of enzymes is very powerful and can apply not only to algorithms requiring extensive precalculation like the FDFT, but also algorithms that are under development, algorithms that are coherent, self-contained entities, or procedures that are highly optimized and therefore require more extensive maintenance.

We have prototyped a class that encapsulates the manipulations required to send an image across the network to a supercomputer facility where highly optimized FDFT procedures can be applied. To the programmer, this class provides the entrance to a tunnel; an image can be shipped into the tunnel and it returns later (but not much later!) out of the tunnel transformed as requested. This facility requires software both on the sending workstation and on the supercomputer. We will be incorporating this facility into a later release of COOL. Other candidate procedures of general interest that might become enzymes are eigenvector and eigenvalue computations and matrix inversion.

Graphlib



Graphlib provides some classes that support representation and processing of computer graphics models and display lists.

Class polyobject

A polyobject is a sequence of Polylines used for representing objects in a computer graphics system. An object of class polyobject is a node in a linked list of polyobjects, each of which contains a name of up to 10 characters, a pointer to a Polyline object that contains the sequence of vertices making up the object, and a pointer to the next polyobject in the sequence.

Constructors

polyobject(const char* name=NULL)
: This constructor creates a polyobject with the indicated name and with null pointers for both the Polyline and the next polyobject.

polyobject(polyobject&)
: The copy constructor duplicates the entire given polyobject by copying the pointers and the name, only. The Polyline is not duplicated.

polyobject(Polyline*, const char* name=NULL)
: This constructor creates a polyobject with the given name pointing to the given Polyline.

polyobject(Polyline*, polyobject*, const char* name=NULL)
: Create a polyobject with the specified name, Polyline pointer and next-polyobject pointer. This constructor is used to place a polyobject pointing to the specified Polyline at the head of a linked list of polyobjects.

~polyobject()
: The destructor deletes the Polyline and passes the delete along to the next polyobject in the list.

Member functions

The first three member function handle the name field of the polyobject.

polyobject& set_name(const char* name)
: Set the polyobject's name to the given string.

char* get_name()
: Return a pointer to the polyobject's name string.

polyobject& print_name()
: Print the polyobject's name to cout.

The next member functions access the pointer members.

polyobject* next()
: Return a pointer to the next polyobject in the linked list. This message is used to advance through the linked list of polyobjects.

Polyline* get()
: Return a pointer to this polyobject's Polyline structure.

`polyobject& attach(polyobject*)`
: Attach the given linked list of `polyobjects` to the end of the linked list headed by `self`.

`polyobject& operator=(polyobject&)`
: Copy the pointers of the argument to `self`.

`polyobject* process(Xform_matrix)`
: Send the given `Xform_matrix` to the current `Polyline` and to the rest of the current linked list of `polyobjects`. This message is used to apply a transformation to all of the `Polyline` data for an object. The return value is a pointer to a new `polyobject` with new `Polylines` that have been so transformed. The current `polyobject` and its `Polylines` are unchanged.

`polyobject& homogenize()`
: To homogenize the representation of an object, homogenize all of the `Polylines` that make it up. Homogenizing involves dividing each point through by its w coordinate.

Commentary

At present, the destructor's design is incompatible with the design of the copy constructor and operator=(). Since the copy and operator= just copy pointers and do not duplicate the entire linked list and the `Polylines`, and since `Polylines` do not have reference counts like `buffers` do, it is not possible to pass a `polyobject` as an argument (but it is permissible to pass a `polyobject*` as an argument!). When the local `polyobject` is deleted, it will delete the `Polyline` that is still needed in the calling scope. Caution is therefore warranted in using the `polyobject`. Never use one as an argument.

Class glob

Glob is an abstract superclass for graphical objects -- either gobjects or groups. A glob contains a name of up to 10 characters, a pointer to the next glob in a linked list of globs, a transformation matrix, and a visibility flag.
It might be desirable, in some unlikely situation, to create a glob directly, so the glob is provided with a hard-coded, displayable representation. The glob also contains a parent pointer that points to the glob that is the parent of this glob's linked list. The parent pointer is important for propagating the effects of transformation matrices through the glob structure.

Constructors

The constructors allow specification of any of the fields of a glob.

`glob(char* name=NULL)`
: Create a glob with the specified name, an identity transformation, a null pointer to the "next" glob, and invisible.

`glob(glob&)`
: Copy all fields of the given glob.

`glob(Xform_matrix, char* name=NULL)`
: Create a glob with the specified name and transformation matrix, a null pointer to the next glob and invisible.

`glob(Xform_matrix, glob*, char* name=NULL)`
: Create a `glob` with the specified name, transformation matrix, and next_glob pointer. This is used to attach the new `glob` as the new head of a linked list of `globs`.

Member functions

`glob& set_name(char* name)`
: Copies the given string into the name field of the `glob`. The name is limited to 10 characters.

`char* get_name()`
: Returns a pointer to the name string of the `glob`.

`glob& print_name()`
: Prints to `cout` the name of the `glob`. If the `glob` is invisible, a # will be prepended to the name.

`virtual void print_structure()`
: Prints to `cout` the name of `self` (using print_name) followed by a space, and forwards the message to the next_glob in the linked list of `globs`.

`glob& attach(glob*)`
: Attaches the given linked list of `globs` to the end of the linked list headed by `self`.

`glob* next()`
: Returns a pointer to the next `glob` in the linked list.

glob* get_parent()
Returns a pointer to the parent glob.

glob& set_parent(glob*)
Sets the parent pointer in self equal to the argument.

glob& set_Xform(Xform_matrix T)
Sets the transformation matrix of self to T.

Xform_matrix get_Xform()
Returns the transformation matrix of self.

glob& print_Xform()
Prints to cout the name of self, the name of the parent of self, and the transformation matrix of self.

virtual void print_Xform_structure(char* name=NULL)
If the name is null, this prints to cout the transformation matrices of all globs in the linked list headed by self. If name is not null, only the named glob(s) will print their transformation matrices. Note that even if self is printed, the message is still passed on through the linked list.

The following group of messages handle modifications of the transformation matrix and propagating orders for such modifications through the linked glob structure. The Xform_type arguments are from the set {TRANSLATE, ROTATE, SCALE}, the axis arguments are from the set {X, Y, Z} and the double arguments are the parameters of the operation. In all cases, the new transformation matrix is created by multiplying the old transformaiton matrix on the LEFT by a transformation matrix containing the indicated operation

glob& transform(Xform_type, axis, double)
Modify the transformation matrix by the indicated 1-axis transformation.

glob& transform(Xform_type, double, double, double=0.0)
Modify the transformation matrix by the indicated 3-axis transformation.

virtual void transform(char*, Xform_type, axis, double)
If the name of self is the same as the first argument, apply the indicated transformation. In any case, pass the message on through the linked list of globs.

virtual void transform(char*, Xform_type, double,double,double)
If the name of self is the same as the first argument, apply the indicated transformation. In any case, pass the message on through the linked list of globs.

virtual polyobject* process(Xform_matrix)
The argument is a viewing transformation matrix that is to be applied to the whole linked list of globs. The result is a polyobject containing a linked list of Polylines produced by transforming the Polylines in the linked list headed by self.

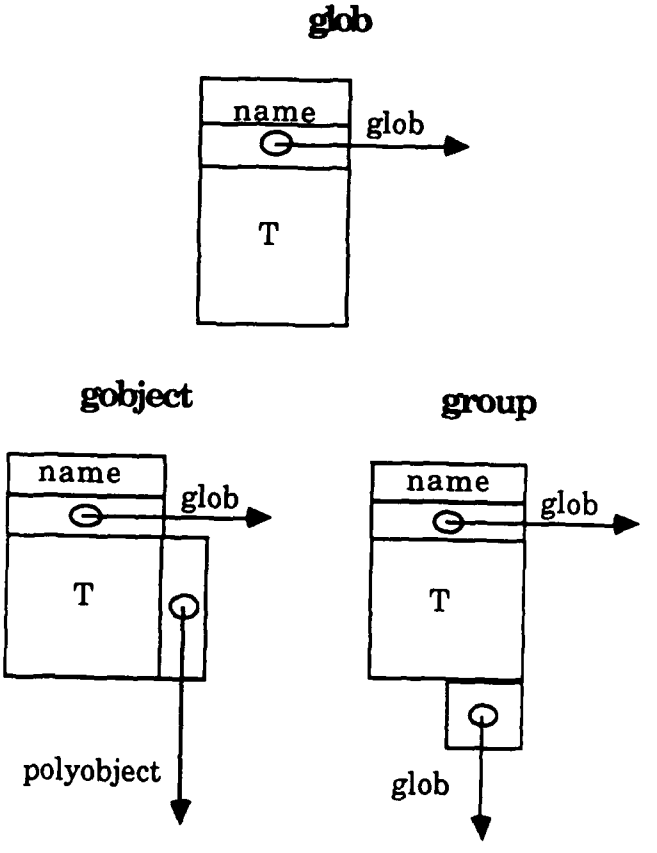
virtual Xform_matrix get_absolute_Xform()
This is a private function that computes the absolute transformation matrix for self by multiplying together all of the transformation matrices in the parent globs all the way to the root of the glob list.

virtual Xform_matrix get_absolute(char*)
Compute and return the absolute transformation matrix for the named glob. (If self is the named glob, invoke get_absolute_Xform().

`glob& visible()`
Set this glob to visible.
`glob& invisible()`
Set this glob to invisible. Invisible is the default for globs.

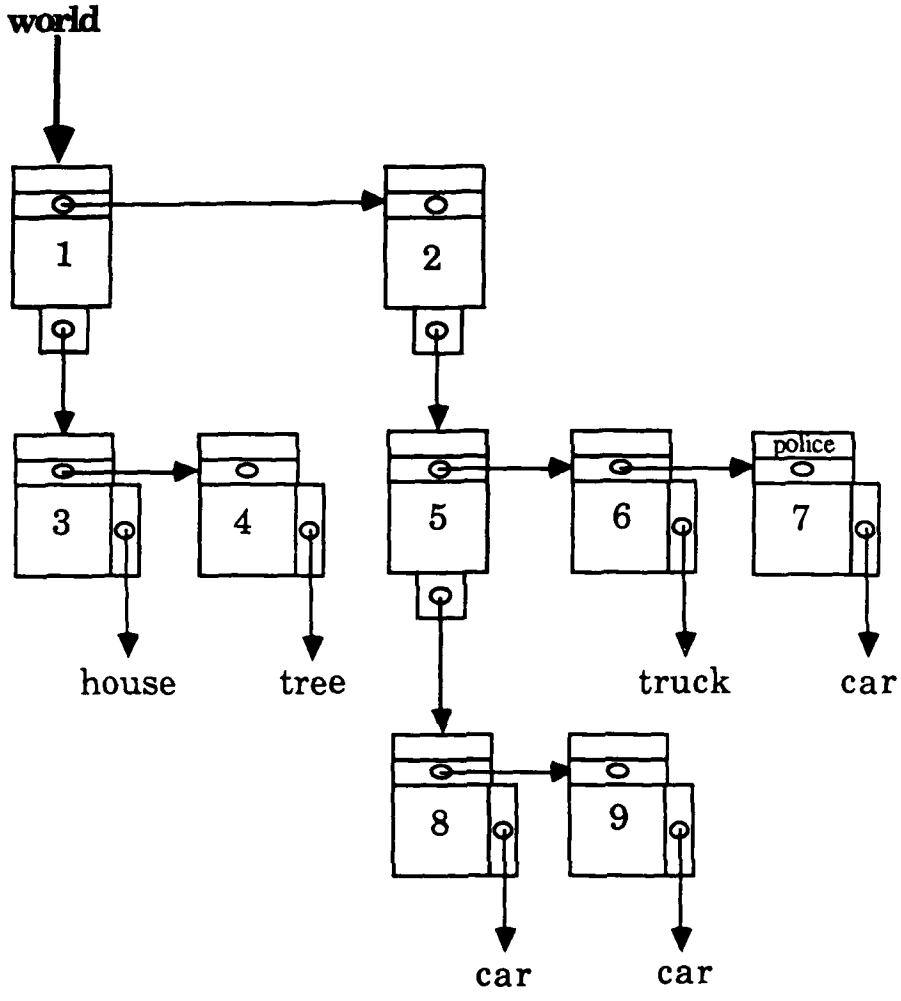
Commentary

The glob hierarchy is one of my best examples of effective inheritance and encapsulation. The diagram below illustrates the structures of these classes.



The diagram shows that `gobject` and `group` inherit all of the data components of `glob` while adding a small extra item that distinguishes them. A `gobject` adds a `polyobject` pointer while a `group` adds a `glob` pointer. The `glob` operations that manipulate the transformation matrix, the name, and the `next_glob` pointer are also inherited from `glob`. What the derived classes must supply is operations on their unique parts and, in a couple of cases, special versions of the operations defined in `glob` that are different due to the additional structure of the derived class.

These classes can be used to create linked display lists of arbitrary complexity, as illustrated below.



In this example, the words "house", "car", "tree", and "truck" refer to `polyobjects` containing graphical representations of the named objects. Notice that in the world model above, there are three instances of "car". Each instance is controlled by a different set of transformation matrices, so the three instances may behave independently. Similarly, the cars and the truck may be moved as a group relative to the house and tree by changing the transformation matrix in node 2. The police car can be moved by itself by sending a message to the world addressed to the police car, for example,

```
world->transform("police",TRANSLATE,X,20.0);
```

This message would modify the transformation matrix in node 7. When a view is to be created, the transformation matrix applied to the car `polyobject` to produce the image of the police car is found by multiplying the viewing transformation matrix by matrices 2 and 7 and processing the car with the result.

It can be difficult to keep up with the viewpoint in an active model. Therefore, it is possible to create a `glob` directly in the model whose purpose is basically to keep up with a viewpoint as the model moves. For example, the viewpoint of the driver of a car might be maintained as a `glob` having a certain fixed relationship to the

car. By asking for the absolute transformation of the viewer `glob`, one can obtain the transformation matrix to apply to the rest of the world. `Globs` are normally invisible, but they can be made visible. They will appear in the model as small eye pyramids. Thus, it is possible to create a viewer `glob` and pan through the scene inspecting the shot before looking at the world from the prearranged view.

Class gobject:public glob

A gobject is a kind of glob that contains a pointer to a polyobject that contains the graphical representation of an object in the virtual world.

Constructors

gobject(gobject&)
: Copy one gobject to another by copying the pointers.

gobject(polyobject*,char* name=NULL)
: Create a gobject with the indicated polyobject and name.

gobject(polyobject*,Xform_matrix,char* name=NULL)
: Create a gobject with the indicated polyobject, transformation matrix, and name.

gobject(polyobject*,Xform_matrix,glob*,char* name=NULL)
: Create a gobject with the indicated polyobject, transformation matrix, name, and next_glob pointer. This constructor is used to add the new polyobject onto the front of a linked list of globs.

Member functions

void print_structure()
: Print to cout the name of self, if any, the name of the polyobject in square brackets, and pass the message to the next_glob.

polyobject* process(Xform_matrix)
: Multiply the given transformation matrix on the LEFT by the one in self and then if self is visible, process the polyobject. Next, return a polyobject* pointing to a list of polyobjects containing the results of processing the rest of the linked list of globs by the given transformation matrix.

Commentary

Class group:public glob

A group is a kind of glob that contains a pointer to another list of globs.

Constructors

group(group&)
: Copy one group node to another by copying the pointers.

group(glob*, char* name=NULL)
: Create a group with the indicated glob list pointer and name.

group(glob*, Xform_matrix, char* name=NULL)
: Create a group with the indicated glob list, transformation matrix, and name.

group(glob*, Xform_matrix, glob*, char* name=NULL)
: Create a group with the indicated glob list, transformation matrix, next_glob pointer, and name. This constructor is used to attach the new group as the head of a linked list of globs.

Member functions

void print_structure()
: Print to cout the name of self and the characters ": (". Next, pass the message down to the model list. Finally, print ") " and pass the message to the next_glob. The result is the structure of the world in a linked list notation.

void print_Xform_structure(char* name=NULL)
: If the name provided is null, print the entire transformation matrix structure of the glob list with self as its head. If a name is provided, and it is the name of self, print the transformation matrix of self and pass the message on to the glob list and to the next_glob.

void transform(char*, Xform_type, axis, double)
: If the first argument is the name of self, apply the indicated transformation to the transformation matrix in self. In any case, send the message on to both the glob list and the next_glob.

void transform(char*, Xform_type, double, double, double)
: If the first argument is the name of self, apply the indicated transformation to the transformation matrix in self. In any case, send the message on to both the glob list and the next_glob.

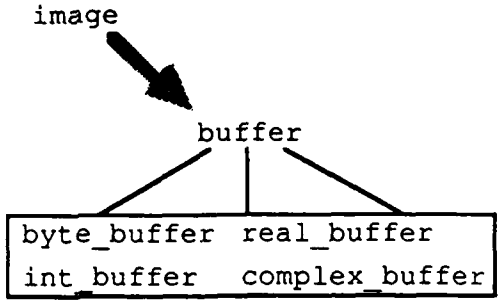
polyobject* process(Xform_matrix)
: Multiply the transformation matrix in self on the RIGHT by the given transformation matrix and send the product in a process message to the glob list. Send the given transformation matrix in a process message to next_glob. Return the concatenation of the results of the process messages sent to the glob list and the next_glob.

Xform_matrix get_absolute(char*)

If `self` is the named `glob`, return the absolute transformation matrix of `self`. Otherwise, pass the message to `next_glob`. If the result is identity, pass the message to the `glob_list` and return the result.

Commentary

Imagelib/Bufferlib



Each image object contains a pointer to a buffer object that is realized as one of the subclasses of buffer (one for each supported type). Messages to image are forwarded to the buffer object where most of the messages are interpreted by the subclass code in a type-dependent manner.

Class image

The image class is the heart of COOL, and yet its function is about the most trivial in the whole library. An image is little more than a pointer to a buffer object. It is not related by inheritance to the buffer class at all; instead, image contains a variable that is of type buffer* and the funciton of most of the messages understood by image is simply to pass the message on to the buffer object. Thus, the image object is very small: one pointer, no matter how big the image is. Thus, image objects may be passed as value arguments without difficulty or inefficiency.

This description of the image class will focus on the external operations while the description of the buffer classes will detail internal operations.

Constructors

Image objects can be created in six ways:

The `null constructor` creates a 1x1 `INTEGER` image.

`image(image& im)`
: Simply copy the `buffer*` in `im`. It does not copy the entire buffer. This definition makes transfer of images as arguments quick and efficient.

`image(int nr, int nc, buffer_type bt)`
: Creates an nrxnc image of the specified buffer type (`BYTE, INTEGER, REAL, COMPLEX`).

`image(subscript_s, buffer_type bt)`
: Creates an nrxnc image of the specified buffer type (`BYTE, INTEGER, REAL, COMPLEX`) using a subscript object to specify the image size.

`image(buffer*)`
: Simply adopts the specified buffer.

`image(image,image,convert_type)`
: Forms a new image as the product of two given images. The type of the new image is the type of the first specified image. The `convert_type` is used if the second argument is complex and the first is not. This constructor is used mainly in filtering operations.

Member functions

The first class of messages return or adjust information about the physical configuration of the image buffer and the window of interest within it.

`buffer* buffer_addr()`
: Simply returns the pointer to the buffer.

`buffer_type type()`
: Returns the type (`BYTE, INTEGER, REAL, COMPLEX`) of the buffer.

`char* storage_addr()`
: Returns a char* pointing to the actual pixel array. This pointer must be cast to the actual storage type of the buffer.

`int total_bytes()`
Returns the number of bytes in the pixel array in the buffer

`int buffer_size()`
Returns the number of pixels in the buffer

`int size()`
Returns the number of pixels in the current window

`subscript shape()`
Returns a subscript holding the size of the buffer and properly initialized internally for use as a **`for`** loop index for manipulating the image.

`image& reset()`
Resets the buffer's window to be the whole buffer.

`image& set_window(subscript ulc)`
Sets the upper left corner of the buffer's window at the indicated `subscript` and its size to be the same as its current size or the rest of the image, whichever is smaller.

`image& set_window(subscript ulc, subscript lrc)`
Sets the window to the indicated upper left corner and lower right corner.

The following set of operations are defined for obtaining and setting individual pixels in an image. The get messages must indicate the type of value requested since member functions are not distinguished by their return value types, but the set messages are overloaded since member functions are distinguished by their argument lists.

With a subscript argument, the subscript is used as an offset within the current image window. With an integer n, the integer is treated as the element number in the buffer's storage array (which stores pixels in row-major order). The convert_type argument determines whether the return value from a complex image is obtained by REAL, IMAGINARY MAGNITUDE or PHASE conversion.

```
int    get_i(int n,     convert_type)
int    get_i(subscript s, convert_type)
double get_r(int n,     convert_type)
double get_r(subscript s, convert_type)
complex get_c(int n)
complex get_c(subscript s)

image& set(int n, int i)
image& set(subscript s, int i)
image& set(int n, double d)
image& set(subscript s, double d)
image& set(int n, complex c)
image& set(subscript s, complex c)
```

The following section consists of arithmetic operations on entire images. Note that the operator=() does not copy pixels, only the buffer pointer. To create a copy, you must use the copy message. The `convert_type` argument is always defaulted to `MAGNITUDE` and is relevant only when operating on a `COMPLEX` image. Allowed values are (`REAL, IMAGINARY, MAGNITUDE, PHASE`).

`image& operator=(image& im)`
Copy the `buffer*` in `im` to the `buffer*` in self. This does NOT duplicate the pixel array.

`image convert(buffer_type to, convert_type)`
Returns a new `image` the same size and shape as `self` but having the specified type.

`init(double d)`
Sets all pixels of an image equal to d.

`image& copy(image& im, convert_type ct=MAGNITUDE)`
Copy the current window in `im` to the current window in `self`.

`image& add(image& im, convert_type ct=MAGNITUDE)`
Add the current window in `im` to the current window in `self`.

`image& sub(image& im, convert_type ct=MAGNITUDE)`
Subtract the current window in `im` from the current window in `self`.

`image& mpy(image& im, convert_type ct=MAGNITUDE)`
Multiply the current window in `im` to the current window in `self`.

`image& div(image& im, convert_type ct=MAGNITUDE)`
Divide the current window in `self` by the current window in `im`.

`image& scale(double add1, double mpy, double add2)`
Compute at each pixel `new=(old+add1)*mpy+add2`

`image& logscale()`
Compute the log (base e) of each pixel. After log scaling you will usually require some kind of contrast stetching since logs are usually small numbers

`image& stats0(double& min, double& max, convert_type)`
Compute the min and max values in the current window.

`image& stats1(double& mean, double& sd, convert_type)`
Compute the mean and standard deviation in the current window

The following section consists of debugging aids of various kinds.

`image& print_pixel(int n, ostream& o=cout)`
Print a particular pixel in the buffer

`image& print_pixel(subscript s, ostream& o=cout)`
Print a particular pixel in the current window

`image& print_window(ostream& o=cout)`
Print the pixels in the current window

Commentary

The image class is the workhorse of COOL. In developing image I learned more about C++ than any other class, and image is the class I most often need to accomplish any research. This is ironic since my final design for image leaves it mainly as a message shuffler that does very little itself.

The window feature is not well-tested.

The decision to make image an intelligent pointer has paid for itself multiple times over in the simplicity of the application code where I don't have to ever worry about pointers to images or passing images as arguments or incompatible storage types or encoded names specific to different types of images.

Class `buffer`

Buffer is an abstract superclass for its type-specific derived classes. The main function of buffer is to maintain some administrative data like the buffer size and shape and to pass all messages down to the derived class.

Constructors

All of the constructors to buffer are protected since buffer is an abstract superclass and should never be created by a user.

A buffer is created by a null constructor, or by specifying the buffer shape desired either by a subscript or by a pair of ints.

The destructor for buffer is a public virtual destructor. It is virtual since the main action required, deletion of the data array, must be handled by the appropriate subclass since we cannot tell here what type the data array is.

Member functions

`Buffer` maintains a reference count that is used by image::~image() to determine whether the destructor for the image class should actually delete the buffer or simply decrease the reference count. The following messages maintain the reference count:

`int ref()`
: Returns the current reference count

`int new_ref()`
: Increments the reference count and returns it.

`int del_ref()`
: Decrements the reference count and returns it. Normally, if the reference count is zero at this point, the destructor should be invoked.

The following messages are concerned with the shape of the buffer and the shape and position of the window within it.

`int size()`
: Returns the number of pixels in the current window

`int buffer_size()`
: Returns the number of pixels in the whole buffer

`subscript_shape()`
: Returns the shape of the buffer with the magic data initialized for use as in a for loop index

`virtual buffer_type type()`
: Returns the type of the buffer, which, of course, must be supplied by the derived class.

virtual int total_bytes()
Returns the total number of bytes in the buffer (from the derived class)
virtual char* storage_addr()
Returns a pointer to the beginning of the pixel data. This pointer must be cast to the approriate type before use.

void reset()
Resets the window to be the size of the whole buffer
void set_window(subscript ulc)
Sets the window so that its upper left corner is at the specified location and its lower right corner is the lower right corner of the buffer.
void set_window(subscript ulc, subscript lrc)
Sets the window to have the indicated upper left and lower right corners

The following get and set functions modify elements of the buffer. If a subscript is given, the subscript is an offset in the current window. If an integer is given, it is used as an absolute offset in the buffer, which is stored in row-major order. The `convert_type` is defaulted to `MAGNITUDE` and controls the mode of conversion from complex to real when required.

```
virtual int    get_i(int, convert_type)
virtual int    get_i(subscript, convert_type)
virtual double get_r(int, convert_type)
virtual double get_r(subscript, convert_type)
virtual complex get_c(int)
virtual complex get_c(subscript)

virtual void set_i(int, int)
virtual void set_i(subscript,int)
virtual void set_r(int, double)
virtual void set_r(subscript,double)
virtual void set_c(int, complex, convert_type)
virtual void set_c(subscript, complex, convert_type)
```

The next group of member functions handles image arithmetic of various kinds.

virtual void init(double)
Set the whole image to the specified value
virtual void copy(buffer*, convert_type)
Copy the window of the specified buffer to the current window in `self`
virtual buffer* convert(buffer_type, convert_type)
Create a new buffer of the specified type to hold the data in the current window, converted according to the specified mode (if self is complex and the desired type is simpler). This message can be used to extract a subimage.

The following arithmetic operations modify `self` to hold the result of the indicated computation.
virtual void add(buffer* b, convert_type)

```
      self=self+b
virtual void sub(buffer* b, convert_type)
      self=self-b
virtual void mpy(buffer* b, convert_type)
      self=self*b
virtual void mpy2(buffer* b, buffer* c, convert_type)
      self=b*c
virtual void div(buffer* b, convert_type)
      self=self/b
```

`virtual void scale(double add1=0.0,double mpy=0.0,double add2=0.0)`
For each pixel p in the window, p=(p+add1)*mpy+add2

`virtual void logscale()`
For each pixel p in the window, p=log(p). Note that you will need to scale after this.

`virtual void stats0(double& min,double& max,convert_type)`
Compute the min and max of the window

`virtual void stats1(double& mean,double& sd,convert_type)`
Compute the mean and standard deviation in the window

`virtual void print_pixel(int,ostream& o=cout)`
Print the value of a pixel to cout.

`virtual void print_pixel(subscript, ostream& o=cout)`
Print the value of a pixel to cout.

`virtual void print_window(ostream& o=cout)`
Print the pixels in the window to cout.

Commentary

Buffer is an abstract superclass, so users should never need to create or manipulate a buffer object. Operationally, messages sent to the buffer class are shunted via the virtual function mechanism to the procedures provided by one of the type-specific subclasses.

The indirection of having messages to image be forwarded to the buffer and interpreted by subclasses gives us a high degree of flexibility and makes possible the definition of type-independent procedures for operating on images. The type-independence is not fully exploited in COOL at present, but the capability to do so is present.

To use the buffer class and its subclasses directly, define a buffer* and assign to it a new object of one of the derived classes. For example,

```
buffer* buf;
buf = new byte_buffer(256,256);
```

Classes byte_buffer, int_buffer, real_buffer, and complex_buffer : public buffer

These classes, all subclasses of `buffer`, contain, manage, and operate upon the pixel data itself. Messages are defined for operating on the data in type-dependent ways. The type dependence is hidden from users by the `buffer` class.

Constructors

The null constructor creates a 1x1 buffer of the appropriate type.
A buffer of the named type can be created with any desired shape by specifying the number of rows and number of columns or by specifying a subscript holding that information.

The destructor deletes the pixel array. The destructor must not be called until the pixel array is ready to be destroyed. See the description of the reference count in `buffer`.

Member functions

The following messages are concerned with the structure of the pixel array.

`buffer_type type()`
: Returns the type of the buffer, which, of course, must be supplied by the derived class.

`int total_bytes()`
: Returns the total number of bytes in the buffer (from the derived class)

`char* storage_addr()`
: Returns a pointer to the beginning of the pixel data. This pointer must be cast to the approriate type before use.

The following get and set functions modify elements of the buffer. If a subscript is given, the subscript is an offset in the current window. If an integer is given, it is used as an absolute offset in the buffer, which is stored in row-major order. The `convert_type` is defaulted to `MAGNITUDE` and controls the mode of conversion from complex to real when required.

```
int    get_i(int, convert_type)
int    get_i(subscript, convert_type)
double get_r(int, convert_type)
double get_r(subscript, convert_type)
complex get_c(int)
complex get_c(subscript)

void set_i(int, int)
void set_i(subscript,int)
void set_r(int, double)
```

void set_r(subscript,double)
void set_c(int, complex, convert_type)
void set_c(subscript, complex, convert_type)

The next group of member functions handles image arithmetic of various kinds.

void init(double)
Set the whole buffer to the specified value
void copy(buffer*, convert_type)
Copy the window of the specified buffer to the current window in `self`
buffer* convert(buffer_type, convert_type)
Create a new buffer of the specified type to hold the data in the current window, converted according to the specified mode (if `self` is `complex` and the desired type is simpler). This message can be used to extract a subimage.

The following arithmetic operations modify `self` to hold the result of the indicated computation.

void add(buffer* b, convert_type)
`self=self+b`
void sub(buffer* b, convert_type)
`self=self-b`
void mpy(buffer* b, convert_type)
`self=self*b`
void mpy2(buffer* b, buffer* c, convert_type)
`self=b*c`
void div(buffer* b, convert_type)
`self=self/b`

void scale(double add1=0.0,double mpy=0.0,double add2=0.0)
For each pixel p in the window, p=(p+add1)*mpy+add2
void logscale()
For each pixel p in the window, p=log(p). Note that you will need to scale after this.

void stats0(double& min,double& max,convert_type)
Compute the min and max of the window
void stats1(double& mean,double& sd,convert_type)
Compute the mean and standard deviation in the window

void print_pixel(int,ostream& o=cout)
Print the value of a pixel to cout.
void print_pixel(subscript, ostream& o=cout)
Print the value of a pixel to cout.
void print_window(ostream& o=cout)
Print the pixels in the window to cout.

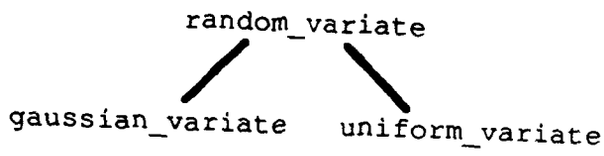
Commentary

The `logscale` message works only on `real_buffer` objects.

Misclib

complex
magpha

histogram
nd_gaussian
subscript
timer



Misclib contains a collection of very useful miscellaneous classes.

Classes `complex` and `magpha`

Classes complex and magpha implement both the real-imaginary and magnitude-phase forms of complex numbers. In both cases the complex values are stored as a pair of `float`s.

Constructors

The null constructors initialize the value to (0,0). Constructors are provided that accept
- a pair of float or double values to be used as the components of the specified representation,
- a variable of the same type (a copy constructor), and
- a variable of the partner type (a conversion constructor).

No destructor is necessary.

Member functions

Member functions for `complex` and `magpha` fall into four categories: data access, simple operations, arithmetic operators, and advanced operations.

The data access functions (identical for both classes) return `float` or `double` values and are as follows:

`double real()`
: Returns the real part

`double imag()`
: Returns the imaginary part

`double mag()`
: Returns the magnitude

`double mag_2()`
: Returns the magnitude squared

`double phase()`
: Returns an angle from $-\pi$ to π)

`double convert(convert_type)`
: Returns a double by applying the specified `convert_type`, which can be `REAL`, `IMAGINARY`, `MAGNITUDE`, or `PHASE` . These symbols are defined in `cool.h`.

Note: The `real` and `imag` messages return `float&` from `complex` and `double` from `magpha`. Thus, they may be used to set components of a `complex`. Similarly, `mag` and `pha` return `float&` from `magpha` and `double` from `complex`.

Simple operations on complex numbers include

`complex conj()`
: complex conjugation

`times_i()`

multiplication by i

recip()
complex reciprocal

exchange(x)
where x is the same class as self and its action is to exchange the values of self and x

print().
Complex objects are printed with syntax (12.3,0.0) while magpha objects are printed with syntax [5.7 exp(1.2i)].

The arithmetic operators defined for both classes are as follows:
= == != unary - * *= / /=
Additional operators defined for complex only are:
+ += binary - -=

The following advanced operations are defined for complex only:
exp() log() sqrt() cos() sin() atan()

Commentary

The complex and magpha classes were among the first classes I wrote in C++. Complex has proven its worth, especially in regard to complex images. Magpha has not been as useful as I envisioned when I wrote it. Perhaps magpha should be made a derived class of complex so that the two classes would be compatible in more situations?

Since they were written so long ago, it would probably be possible now to shorten both class definitions by eliminating redundant messages. For example, having three versions of operator=() to cover assignments of magpha, complex, or double values is probably massive overkill. The C++ translator is now more intelligent about conversions than it was when I started using the language, and the automatic type system might be sufficient now to handle these conversions based on only one version of operator=().

The value of inlining functions could be evaluated by timing experiments on large images stored in complex form. The massive number of repeated operations required for example in scaling or FFT computations would probably reveal just what we do or do not gain by using inline functions.

The complex and magpha definitions depend on each other directly. This mutual dependency is handled by a forward class definition inserted in the .d files of both classes (though only one such insertion, for the class whose .h file is read first, is really required). Since complex is the first of these class definitions to be read during compilation the constructor that creates a complex from a magpha can not be inlined: since the magpha class definition is not yet entered, the compiler does not know what operations on magpha objects are available. Thus, the complex(magpha&) constructor and the operator=(magpha&) functions are the only simple member functions in these two classes that are not inlined. Should one reverse the order in which these class definitions are read during

compilation, syntax errors will result until the functions in `magpha` that need to know the internal structure of `complex` are removed from the `.h` file.

Class histogram

The histogram class computes histograms and cumulative histograms of images and patterns. A histogram is defined by three parameters: low, high, and delta. Low and high are the extreme values covered by the histogram. Any values less than low are counted as underflows; any values greater than high are counted as overflows. The histogram bins are of size delta. The number of bins is computed from low, high, and delta and that many integer values (plus two to count the overflows and underflows) are allocated in the histogram buffer.

Constructors

A copy constructor and a minimal destructor are provided. The rest of the constructors are as follows:

histogram(double low, double high, double delta)
: Allocates the required memory and sets the buffer equal to zero. I have yet to discover an interesting use for it.

histogram(image im, double low, double high, double delta, convert_type ct)
: Computes the histogram of an image. The parameters default to the following values: low=0.0, high=255.0, delta=1.0, and ct=MAGNITUDE. The ct parameter is used if the image is stored in complex form and is ignored otherwise.

histogram(pattern& p, double low, double high, double delta)
: Computes the histogram of values in a pattern. The parameters default to the following values: low=0.0, high=100.0, delta=1.0.

Member functions

The member functions fall into three categories: data access, computation, and presentation.

Data access functions available for histogram are as follows:

int size()
: Returns the number of bins in the histogram, including the underflow and overflow bins.

int* storage_addr()
: Returns a pointer to the bin array beginning with the underflow count.

int& operator[](int i)
: Returns the ith bin count, where 0 is the underflow count and entry size()-1 is the overflow count.

histogram& info(double& low, double& high, double& del, int& size, int& under, int& over)
: Returns about everything you would need to know in one call. Note that the data is returned in the arguments.

Computational member functions are as follows:

`histogram& operator=()`
: Assigns the values of one histogram to another.

`int n()`
: Returns the number of items entered into the `histogram`, including overflows and underflows.

`int max_value()`
: Returns the maximum count in any bin, excluding the underflow and overflow bins

`void mode(int& freq, double& value)`
: Returns in the arguments the number of elements in the bin and the lower bound on the bin having the largest number of entries, underflow and overflow bins excluded.

`double median()`
: Returns the lower bound of the bin containing the n/2 entry in the `histogram`, counting from the underflow bin.

`void stats1(double& mean, double sd)`
: Returns the mean and standard deviation of the values in the histogram, underflow and overflow excluded. The effect is to fit a normal distribution to the histogram. The values are computed using bin numbers and then converted to indicate the lower bound of the mean bin and the deviation in the scaled units.

`histogram cumulative()`
: Returns a new `histogram` with the same parameters as the current one, but the bin counts are the cumulative bin counts of the current `histogram`. The accumulation includes the underflow bin.

Two presentation messages are provided, as follows:

`histogram & print(ostream& o)`
: Prints the parameters and bin values to an output stream. The output stream is defaulted to `cout`.

`polyobject* plot()`
: Returns a `polyobject*` containing a `Polyline` holding the vertex coordinates of a plot of the `histogram` fitting into the unit square with corners at (0,0) and (1,1). Thus, this plot is ready to send to the display devices defined in `displaylib`.

Commentary

The `histogram` class works all right, but there is little else to recommend it. The printouts are pretty crummy and the plots are mediocre. It would not be difficult to improve this class, but still, *it does work*.

In the `/usr/image` library developed at UNC-Chapel Hill, the histogram of an image is stored on disk with the image. I still believe that that was a terrible mistake. I strongly prefer treating the histogram as a separate object from the image it describes having its own uses and representations.

It is not clear whether some of the values being returned from the computational functions are well defined. An option to consider if for all of the computational functions to return bin numbers and have another message to convert a bin number to a minimum or midpoint of a bin's range.

Class nd_gaussian

The nd_gaussian class implements an n-dimensional Gaussian density. The mean vector and covariance matrix may be specified by the user. This density can be spatially sampled, the Mahalanobis distance of a point to the density may be computed, and the nd_gaussian can be used as the probability density for a random number generator.

Constructors

The default constructor creates a 2D Gaussian with zero mean vector and identity covariance matrix. A copy constructor is provided.

nd_gaussian(int d)
: Creates a d-dimensional Gaussian with zero mean vector and identity covariance matrix.

nd_gaussian(pattern& p, sq_matrix& m)
: Creates a Gaussian with the same dimensionality as the length of p and with a covariance matrix as given by m. An error abort occurs if both dimensions of m are not equal to the length of p. The arguments are not modified by this constructor.

Member functions

pattern get_mean()
: Returns the mean vector as a pattern.

sq_matrix get_covariance()
: Returns the covariance matrix as a sq_matrix.

double normfactor()
: Returns the real normalizing constant

$$\frac{1}{2\pi^{d/2}|\Sigma|^{1/2}}$$

where d is the dimensionality of the nd_gaussian, Σ is the covariance matrix, and $|\Sigma|$ denotes the determinant of Σ.

nd_gaussian& set_mean(pattern& p)
: Sets the mean vector of the nd_gaussian to p. The argument is not changed.

nd_gaussian& set_covariance(sq_matrix& m)
: Sets the covariance matrix to m. The argument is not changed.

double z_dist(pattern& p)
double operator-(pattern& p)
: Return the Mahalanobis distance of p from the nd_gaussian. This distance is computed according to the formula

$$[(p-\mu)\Sigma^{-1}(p-\mu)^T].$$

`pattern generate()`
`pattern operator>>(pattern& p)`

Return a `pattern` randomly generated according to the density given by the `nd_gaussian`. The `operator>>` also returns the generated `pattern` in its argument, `p`.

Commentary

The `nd_gaussian` class is an effective encapsulation, hiding away many nettlesome details that could distract one from more important tasks. The set of operations available now is minimal but it has sufficed for the uses I have made of the class.

I suspect that the current method for generating random patterns from the `nd_gaussian` is buggy.

Also, the results for dimensionalities greater than 3 should not be trusted until the inverse functions in the `matrix` side of the linear algebra tree are verified (and since they are probably wrong, corrected).

Class subscript

The subscript class encapsulates the nature of indexes to multidimensional arrays and is used with images and in the 2D side of the linear algebra tree. The subscript class provides a variety of constructors and is optimized for fast execution as the controlling index of for loops. Functions for obtaining the subscript of neighbors of a location are also provided. Subscript also contains the critical code for implementing rectangular regions-of-interest in images and matrices.

Subscripts contain a set of internal variables (I will call them "magic values" later) that are used to implement regions of interest and fast indexing for for loops. These internal variables must be initialized correctly with respect to the array or buffer to be indexed. They are the maximum allowed values of the subscript indices, the offset into the matrix of the upper left front corner of the region of interest, and the true row dimension of the full array. A subscript for manipulating a 256x256 image must know the true size of the array it is indexing in, so the subscript must be constructed in such a way as to initialize the magic values. For example, subscript s(0,0,im.shape()); would do the job. So would s = subscript(0,0,subscript(256,256)); and other forms as well.

Constructors

Null and copy constructors are provided. The null constructor initializes the subscript to (0,0) and the copy constructor copies everything including the magic values.

The following forms of the constructor are available:

initial value maximum offset

```
subscript::subscript(int, int)
subscript::subscript(int, int, subscript)
subscript::subscript(int, int, subscript, subscript)
subscript::subscript(subscript)
subscript::subscript(subscript, subscript)
subscript::subscript(subscript, subscript, subscript)
```

The first argument(s) specify the initial value of the subscript; the subsequent arguments specify the maximum values of the subscript and the offset of the subscript in its array. The maximum field of the subscript given as the maximum defines the actual array size unless it is less than the specified value of the maximum, in which case the specified value is taken as the actual array size.

Member functions

`int r()`

Return the current column subscript

`int c()`
Return the current row subcript

`int size()`
Return the product of the row and column values. This is intended as the size of the ROI above and left of the indicated location.

`int maxsize()`
Returns the product of the row and column of the max entry. This is the number of elements in the whole array (not the ROI).

`subscript get_max()`
Returns a `subscript` whose value is the max entry of the current `subscript`.

`subscript& print(ostream&)`
Prints the `subscript` to the `ostream`, which is defaulted to `cout`. The syntax is [5, 12].

The next group of member functions concern the conversion of the `subscript` into an integer array index.

`int index(subscript& max, subscript& off)`
Returns the index using the `subscript`'s own index value and the supplied max and offset values.

`int index(subscript& dim)`
Returns the index using the given max and an assumed 0 offset.

When using the `subscript` as the controlling variable of a `for` loop, the following member functions should be used:

`void init()`
Initializes the `subscript` to (0,0) and prepares the magic values for fast operation in the `for` loop.

`int test()`
Returns 1 if you have finished passing through the array, 0 otherwise.

`subscript& operator++()`
Modifies the magic values to point to the next item in row-major order and makes appropriate corrections for end-of-row and end-of- array.

`int index()`
Returns the index precomputed by the above functions.

To use a `subscript` to control a `for` loop, the `for` loop structure should be

```
for(s.init();s.test();s++)
```

Inside the loop, the value of the index is obtained by `s.index()`. Inside the loop, the `s.r()` and `s.c()` functions still give correctly the current row and column coordinates.

A variety of assignment and arithmetic operations are defined on subscripts.

`subscript& set_to_max()`
Sets the subscript's current value equal to its max entry.

`subscript& operator=(subscript&)`
Copies all parts of the `subscript`, including the magic values.

Several `set` functions are available with argument sequences as follows:
`subscript& set(int, int)`
`subscript& set(int, int, subscript)`
`subscript& set(subscript)`
`subscript& set(subscript, subscript)`
As before, the second subscript argument sets the maximum values of self.

Arithmetic operators defined on subscripts are as follows:
The += and -= operations modify the current value of self without changing the magic values.
`subscript& operator+=()`
`subscript& operator-=()`

The + and - operations create a new `subscript` with the magic values of `self`.
`subscript operator+()`
`subscript operator-()`

`subscript min(subscript)`
Returns a new subscript containing in each position the minimum of the corresponding position in the argument and self.
`subscript max(subscript)`
Is analogous for the maximum values.
`subscript clamp()`
Is used to ensure that a `subscript` is in its bounds. If an entry is less than zero, `clamp()` sets it to zero. If the entry is greater than its max allowed value, it is set to the max allowed value.

Finally, there is a series of functions that return `subscripts` of the neighbors of the currently indexed location. The names encode what the function does to the two indices: `nul` is nothing, `dec` is decrement and `inc` is increment. If we assume the `subscripts` are interpreted as (row,col) then their action is as follows:

function	returned location
`incinc()`	below right
`incdec()`	below left
`decinc()`	above right
`decdec()`	above left
`nulinc()`	right
`nuldec()`	left
`incnul()`	below
`decnul()`	above

Commentary

`Subscript` strikes me at different times as either one of my most elegant abstractions or as one of the biggest kludges in the library. By encapsulating the

notion of "array indexes" in this class, I have, in principle, removed indexing concerns from all kinds of images and arrays. In addition, when I created this abstraction, the ability to handle rectangular regions of interest - rectangular subregions of an array - fell out almost for free! And the mechanism for handling `for` loops looks like a big win both in code complexity and in speed of execution.

Unfortunately, it seems that I am still bound far too closely to the notion of two-dimensional arrays and 2D `subscript`s. I keep running into trouble when I try to extend the notion of `subscript` to higher dimensions, especially in regard to neighborhoods. The biggest kludge in `subscript` is `incdec`, `decinc` and all that trash. Send me your ideas.

I have found little use for the arithmetic operations on `subscript`s. I use the `for` loop constructs every day.

To use the `for` loop constructs correctly, the `subscript` used as the `for` loop index must know about the real size of the `buffer` and the size of the region of interest in the `buffer`. How does it get this information? By asking the `buffer` (or, equivalently, the `image`)! Assuming that `im` is an `image`, I have used the constructions

```
subscript s(im.shape());
```

and

```
s=im.shape or subscript s=im.shape
```

equally well. The key is that the `subscript` returned by `image::shape()` has all of the necessary magic values defined for that `image`. Now, if one changes the `image` size or the parameters of the region of interest, then the index `subscript` is no longer valid, but it can be updated by another call to `im.shape()`.

Class timer

`Timer` provides basic timing facilities suitable for profiling programs.

Constructors

A null constructor is provided that sets the `timer` to `OFF` with no time accumulated and with no name.

The constructor `timer(char* name)` creates a timer with no time accumulated, in the `OFF` state, and with the name specified (up to 15 characters).

Member functions

`timer& set_name(char* name)`
: Sets the name of the timer to the given character string (15 character limit).

`timer& start()`
: Records the current time for use as the beginning of an interval to be timed.

`timer& stop()`
: Records the current time as the end of the current interval. The difference between the time recorded by `start()` and the current time is added to the elapsed time accumulator for this timer.

`timer& reset()`
: Sets the accumulated time for this `timer` to 0. The run state (`RUNNING` or `STOPPED`) of the timer is not changed.

`timer& report()`
: Prints to `cout` the state of the timer, including its name, its run state, the number of intervals recorded, and its cumulative elapsed time.

Commentary

The `timer` operates on clock elapsed time only, not CPU time. Either adding an option to select CPU time or simply incorporating it in the operation of the class would be welcome.

This is a nice, effective encapsulation of a multitude of details involving UNIX system procedures and `.h` file inclusions. I have used this class to profile procedures in a long program. Use of the class was made easier by allocating an array of `timers` and then defining macros to turn them on and off:

```
#define ON(num)   {timers[num].start();}
#define OFF(num)  {timers[num].stop();}
```

Invocations of these macros were distributed through the program at appropriate critical points.

The present `timer` class has no copy constructor or `operator=()` and so it cannot be passed as an argument to a procedure. I can't think of why I would want it to

be - profiling tools should themselves keep a low profile, so I find the strategy of making the timers global to be effective and acceptable.

I considered writing the `timer` class to hold an array of timers, but I found that declaring the array of `timers` in the program being profiled was simple enough, especially with the macros above, and of course, the `timer` class is simpler as just one timer.

Class `random_variate`

`Random_variate` is an abstract superclass providing services to its subclasses concerned with producing variates with different probability distributions.

Constructors

The protected constructor `random_variate()` is invoked by the subclass constructors.

Member functions

`u01_rv()`
: This is a protected member function that uses the UNIX `random()` function to produce a uniformly distributed random number in the range (0..1).

`void new_seed(int)`
: Provides a new random number seed to the UNIX `random()` function. If the argument is nonzero, the argument is used as the new seed. If the argument is zero (the default) then a new seed is generated from the UNIX internal clock. This automatically generated seed value is not available to the programmer, so if a reproducible sequence of random numbers is desired, the seed should be provided explicitly.

Commentary

`Random_variate` is an example of effective information hiding. This class encapsulates (hides) the operation of the UNIX random number generator in a manner that can be shared by subclasses. It does no more than that, but in doing so, it hides the usage of the timer to generate seeds, the `random()` and `srandom()` functions in the UNIX system, and the value of `BIGNUM`, which is the largest representable integer, used to convert the integer output of `random()` to a `double` in the range (0..1).

Classes uniform_variate and gaussian_variate :public random_variate

These classes are subclasses of random_variate used to generate random numbers with uniform or gaussian distributions, respectively.

Constructors

Both classes have public constructors that take two doubles. For uniform_variate, the arguments are interpreted as the high and low limits of the uniform density. The arguments are defaulted to 0.0 and 1.0. For gaussian_variate, the arguments are interpreted as the mean and standard deviation of the desired distribution. The arguments are defaulted to 0.0 and 1.0 (a standard Normal density).

Member functions

Note that the function new_seed(int) is inherited from random_variate.

Integer or real values may be obtained either by a member function call or by using operator>>. Integer values are obtained by truncating the double value computed by the value() function. The member functions available are:

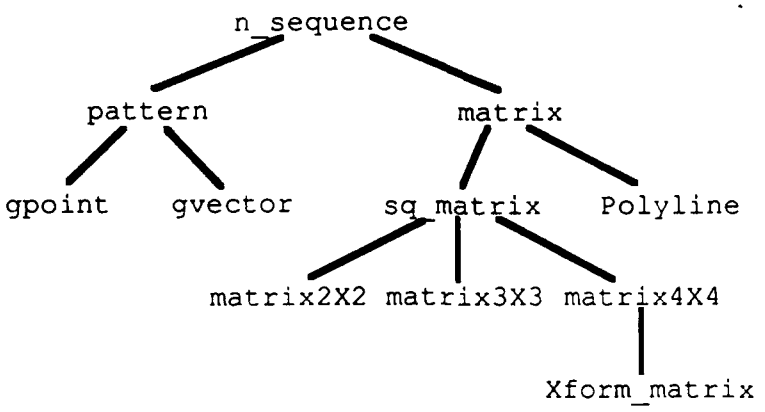
```
double value()
int    ivalue()
int    operator>>(int)
double operator>>(double)
```

The operators both set their arguments equal to the result and return the result as the value of the expression.

Commentary

The value returned for a uniform_variate is low+U(0,1)*(high-low), where U(0,1) is a uniformly distributed random variate on the interval (0,1) provided by the superclass. The value returned for a gaussian_variate is mean+G(0,1)*sd where G(0,1) is a standard Normal random variate computed in gaussian_variate::value() from U(0,1) random variates using a well-known algorithm.

Nseqlib



Nseqlib defines classes of 1D and 2D numerical sequences, including 1D arrays, points and vectors, and various kinds of matrices. The integration of these classes through the inheritance hierarchy is of particular interest. This hierarchy is called the Linear Algebra Tree (LAT).

Class n_sequence

A numerical sequence is a 1D array of doubles with some minimal operations defined on them.

Constructors

n_sequence()
: Construct an n_sequence of length 1.

n_sequence(n_sequence& ns)
: Duplicate the given n_sequence into self.

n_sequence(int i)
: Construct an n_sequence of length i.

~n_sequence()
: Delete the data array.

Member functions

int size()
: Return the number of elements in the n_sequence.

char* storage_addr()
: Return a pointer to the data array that must be cast to double before use.

n_sequence& zero()
: Set all elements of the data array to zero.

n_sequence& copy(n_sequence& ns,int offsetns=0,int offset=0)
: Copy elements of ns into self, beginning with element offsetns of ns and with the offset element of self. Elements are copied until the end of either array is reached. The size os self remains unchanged.

n_sequence& replace(n_sequence& ns)
: The data array in self is deleted and a new array is created and assigned to self that is identical with that in ns.

The following are element-by-element operations on the data array of self. Notice that self is changed by these operations.

n_sequence& negate()
: Negate each element of self

n_sequence& scale(double)
: Multiply each element of self by the indicated scalar

n_sequence& add(n_sequence&)
: Add the each element of the given n_sequence to the corresponding element of self. Stop when either array ends.

n_sequence& sub(n_sequence&)
: Subtract each element of the given n_sequence from the corresponding element of self. Stop when either array ends.

n_sequence& print(int number_to_print, ostream& o=cout)

Print to the indicated `ostream` (`cout` by default) the indicated number of elements of `self`.

`int operator==(n_sequence&)`
Determine whether the given `n_sequence` is equal to `self`. To be equal, the sizes must be equal, then the contents must be equal.

`n_sequence& operator=(n_sequence& ns)`
The data array in `self` is deleted and a new array is created and assigned to `self` that is identical with that in `ns`.

`double& operator[](int index)`
Return a reference to the indicated element of `self`. Note that since a reference is returned, modification to `n_sequence` values can be performed using this message.

Commentary

Class pattern:public nsequence

A pattern is a one dimensional array of values that can be interpreted as the coordinates of a point or vector in a multidimensional space. It differs from an n_sequence mainly in the set of operations available.

Constructors

pattern()
: Create a pattern of length 1.

pattern(int s)
: Create a pattern of length s.

pattern(pattern& p)
: Duplicate pattern p into self.

pattern(subscript& s)
: Create a pattern of length 2 initialized to the value of s.

Member functions

double length()
: Return the length of the pattern (root sum of squares of feature values, or square root of dot product with itself).

pattern normalize()
: Return a new pattern equal to the product of self and the reciprocal of its length.

pattern transform(matrix& m)
: Return a new pattern equal to the product of self and the matrix: form dot products of self with the columns of m. The length of self must equal the number of rows of m. The length of the result is the number of columns of m.

double dotprod(pattern&)
: Return the dot product of self and the argument

double simpson(double width=1.0)
: Return the integral of the curve defined by the pattern, where the interval between features is width/(size-1) and the integral is computed using Simpson's Rule. The size must be odd.

pattern& operator=(pattern& p)
: Delete the data array and replace it with a new one having the same size and content as the argument.

pattern& operator=(subscript& s)
: Delete the data array and replace it with an array of size 2 having the value of the argument.

The following messages return the result of a computation on self, and do not modify self.

pattern operator-()
: Return a new pattern the same size as self with the values negated.

pattern operator+(pattern& p)
: Return a new pattern computed as the sum of self and p. If p is shorter than self, the extra elements of self are unchanged; if p is longer, the extra elements of p are ignored.

pattern operator-(pattern& p)
: Return a new pattern computed as self-p. If p is shorter than self, the extra elements of self are unchanged; if p is longer, the extra elements of p are ignored.

pattern operator*(double d)
: Return a new pattern computed by multiplying through self by d.

double operator*(pattern& p)
: Return the dot product of self and p.

pattern operator*(matrix& m)
: Return a new pattern resulting from multiplying self by the matrix.

Commentary

More member functions are needed to support summation of the elements of a pattern and missing data flags.

Class gpoint:public pattern

Gpoint implements points in homogeneous 3-space for use in computer graphics, inheriting many of its operations from `pattern`. `Gpoint` differs from `gvector` in the operations that are allowed and the desired effects of some of the operations. Formally, the `w` coordinate of a `gpoint` is always equal to 1.0.

Constructors

`gpoint()`
Construct the origin.
`gpoint(gpoint& p)`
Copy the argument.
`gpoint(gvector& v)`
Copy the vector's coordinates and make it a point.
`gpoint(subscript& s)`
Copy the subscript's value into the x and y components
`gpoint(double x, double y, double z=0.0)`
Set the components as indicated.

Member functions

`double x()`
Return the x coordinate.
`double y()`
Return the y coordinate.
`double z()`
Return the z coordinate.
`gpoint& set(double x, double y, double z=0.0)`
Set the coordinates as indicated.
`gpoint& homogenize()`
If the w coordinate is not 1, divide through by w.
`double distance()`
Compute the distance of the gpoint from the origin. Don't use the homogeneous coordinate w in the distance computation.
`double distance(gpoint& p)`
Compute the distance from `self` to `p`.
`gpoint& reflect()`
Set `self` to its reflection about the origin (negate x, y, and z).
`gpoint& reflect(gpoint& p)`
Set `self` to its reflection about `p`.
`gpoint& translate(double x, double y, double z)`
Translate `self` as indicated.
`gpoint& translate(gvector&v)`
Add the vector argument to `self`.
`gpoint& print();`
Print `self` using syntax [1.0,2.0,3.0].
`gpoint& operator=(gpoint& p)`

Set `self` equal to `p`.

`gpoint& operator=(gvector& v)`

Copy the x, y, and z coordinates of the `gvector` into `self`.

`gpoint operator-()`

Return a new `gpoint` with negated x, y, and z coordinates.

`void operator+(gpoint&)`

Print an error message and quit.

`gpoint operator+(gvector& v)`

Return a new `gpoint` resulting from adding the coordinates of `self` and `v`.

`gvector operator-(gpoint& p)`

Return a new `gvector` resulting from subtracting `p` from `self`.

`gpoint operator-(gvector& v)`

Return a new `gpoint` resulting from adding `gvector v` to `self`.

`gpoint operator*(matrix& T)`

Return a new `gpoint` resulting from transforming `self` by `T`.

`gpoint operator*(double d)`

Return a new `gpoint` resulting from multiplying the x, y, and z coordinates of `self` by `d`.

Commentary

Class gvector:public pattern

Gvector implements vectors in homogeneous 3-space for use in computer graphics, inheriting many of its operations from `pattern`. `Gvector` differs from `gpoint` in the operations that are allowed and the desired effects of some of the operations. Formally, the `w` coordinate of a `gvector` is always equal to 1.0.

Constructors

`gvector()`
Construct a zero vector.
`gvector(gvector& v)`
Construct a copy of v.
`gvector(gpoint& p)`
Construct a vector with the same x, y, and z as p.
`gvector(subscript& s)`
Construct a 2D vector with x= column and y=rows
`gvector(gpoint& p, gpoint& q)`
Construct the vector `p-q`.
`gvector(double x,double y,double z=0.0)`
Construct the indicated vector.

Member functions

`double x()`
Return the x coordinate
`double y()`
Return the y coordinate
`double z()`
Return the z coordinate

`gvector& set(double x, double y, double z=0.0)`
Set `self` as indicated.

`gvector normalize()`
Return a new vector computed by multiplying through `self` by the reciprocal of its length.
`gvector crossprod(gvector&)`
Return a new vector equal to the cross product of `self` and the argument.
`gvector& print()`
Print `self` to `cout` with syntax <1.0,2.0,3.0>.
`gvector& operator=(gvector& v)`
Copy the elements of v into `self`.
`gvector& operator=(gpoint& p)`
Copy the x, y, and z coordinates of `p` into `self`.
`gvector operator-()`
Return a new `gvector` that is the negative of `self`.

gvector operator+(gvector& v)
Return a new gvector that is the sum of self and v.
gpoint operator+(gpoint& p)
Return a new gpoint that is the sum of self and p.
gvector operator-(gvector& v)
Return a new gvector that is equal to self minus v.
void operator-(gpoint&)
Print an error message and halt. This is illegal.
gvector operator*(double d)
Return a new gvector equal to self scaled by d.
gvector operator*(matrix& T)
Return a new gvector resulting from multiplying self by matrix T.
double operator*(gvector& v)
Return the dot product of self and v.
gvector operator%(gvector& v)
Return a new gvector equal to the cross product of self and v.

Commentary

Class `matrix:public n_sequence`

A matrix is an n_sequence that knows about rows and columns.

Constructors

`matrix()`
Create a 1x1 matrix of zero.
`matrix(matrix& m)`
Create a duplicate of m.
`matrix(subscript s):(s.size())`
Create a `matrix` shaped as indicated in the value of `s`.
`matrix(int r, int c):(r*c)`
Create a `matrix` shaped as indicated: r rows and c columns.

Member functions

`subscript shape()`
Return the shape of `self`.
`pattern get_row(int)`
Return a `pattern` set to the length and content of the specified row of `self`.
`matrix& set_row(int, pattern&)`
Set the specified row of `self` equal to the specified `pattern`.
`pattern get_col(int)`
Return a `pattern` set to the length and content of the specified column of `self`.
`matrix& set_col(int, pattern&)`
Set the specified column of `self` equal to the given `pattern`.
`matrix& replace(matrix& m)`
Delete the data array and recreate it to be a duplicate of m's data array.
`matrix multiply(matrix&)`
Return a `matrix` computed as the product of `self` and the argument
`matrix transpose()`
Return a new `matrix` equal to the transpose of `self`.
`matrix& print(ostream& o=cout)`
Print `self` to cout.
`matrix& operator=(matrix& m)`
Delete the data array and recreate it to be a duplicate of m's data array.
`matrix operator-()`
Return a new `matrix` in which each element of `self` is negated.
`matrix operator-(matrix&)`
Return a new `matrix` computed as `self` minus the argument.
`matrix operator+(matrix&)`
Return a new `matrix` computed as the sum of `self` and the argument.
`matrix operator*(double d)`
Return a new `matrix` in which each element of `self` is multiplied by `d`.
`matrix operator*(matrix& m)`

Return a new `matrix` equal to the product of `self` and m.

`matrix operator!()`

Return a new `matrix` equal to the transpose of `self`.

`matrix& operator-=(matrix&)`

Subtract the argument from `self`.

`matrix& operator+=(matrix&)`

Add the argument to `self`.

`matrix& operator*=(double d)`

Multiply each element of `self` by the constant.

`matrix& operator*=(matrix& m)`

Multiply `self` by m in-place.

`double get(subscript s)`

Return the indicated element of `self`.

`double get(int r,int c)`

Return the indicated element of `self`.

`matrix& set(subscript s,double d)`

Set the indicated element of `self` to the given value.

`matrix& set(int r,int c,double d)`

Set the indicated element of `self` to the given value.

`double& operator()(int, int)`

Return a reference to the indicated element of `self`. Note that this allows the element to be changed directly by the application.

Commentary

Class sq_matrix:public matrix

A sq_matrix is a matrix having an equal number of rows and columns. Some additional operations can be performed on such matrices.

Constructors

sq_matrix()
: Construct a 1x1 matrix of zero.

sq_matrix(sq_matrix& m)
: Duplicate the given matrix.

sq_matrix(int s, int t=0)
: Create an sxs `matrix` initialized to zero if t=0 and to an identity matrix if t=1.

Member functions

int dimension()
: Return the dimension of `self`.

sq_matrix& identity()
: Set `self` to an identity matrix.

sq_matrix minor_matrix(int r, int c)
: Create a new `sq_matrix` of dimension one less than `self` containing all of `self` except the rth row and the cth column.

double determinant()
: Return the determinant of `self`.

sq_matrix& inverse()
: Invert `self` in-place.

sq_matrix& operator=(sq_matrix& m)
: Make `self` a duplicate of m.

sq_matrix& operator=(matrix& m)
: Make `self` a duplicate of m. The dimensions of m must be equal.

sq_matrix operator-()
: Return a new `sq_matrix` in which the elements of `self` are negated.

sq_matrix operator+(sq_matrix&)
: Return in a new `sq_matrix` the sum of `self` and the argument.

sq_matrix operator-(sq_matrix&)
: Return in a new `sq_matrix` `self` minus the argument.

sq_matrix operator*(double)
: Return a new `sq_matrix` with each element of `self` multiplied by the argument.

sq_matrix operator*(sq_matrix&)
: Return a new `sq_matrix` equal to the matrix product of `self` and the argument.

matrix operator*(matrix&)
: Return a new `matrix` equal to the product of `self` and the argument.

sq_matrix& operator+=(sq_matrix&)
: Add the argument to `self`. The sizes must agree.

`sq_matrix& operator-=(sq_matrix&)`
Subtract the argument from `self`. The sizes must agree.
`sq_matrix& operator*=(double)`
Multiply each element of `self` by the argument.
`sq_matrix& operator*=(sq_matrix&)`
Multiply `self` by the argument.
`sq_matrix operator!()`
Return a new `sq_matrix` equal to the transpose of `self`.
`sq_matrix operator~()`
Return a new `sq_matrix` equal to the inverse of `self`.

Commentary

We need better implementations of determinant and inverse. Also, computation of eigenvectors and eigenvalues are needed.

Classes `matrix2X2, matrix3X3, and matrix4X4` `:public sq_matrix`

These classes are defined to allow optimized operations for these small matrices. These classes accept identical messages, so only `matrix2X2` will be described below.

Constructors

`matrix2X2()`
Construct a 2x2 zero matrix.

`matrix2X2(int typ)`
If typ=0, construct a 2x2 zero matrix. Otherwise construct a 2x2 identity matrix.

`matrix2X2(matrix2X2& m2)`
Duplicate the given matrix.

Member functions

`matrix2X2& identity()`
Make `self` an identity matrix.

`matrix2X2& zero()`
Make `self` a zero matrix.

`double determinant()`
Return the determinant of `self`.

`matrix2X2& inverse()`
Invert `self` in-place.

`matrix2X2& operator=(matrix2X2&)`
Make `self` a duplicate of the argument.

`matrix2X2& operator=(sq_matrix&)`
Make `self` a duplicate of the argument. Sizes must agree.

`matrix2X2 operator-()`
Return a new `matrix2X2` with each element of `self` negated.

`matrix2X2 operator+(matrix2X2&)`
Return in a new `matrix2X2` the sum of `self` and the argument.

`matrix2X2 operator-(matrix2X2&)`
Return in a new `matrix2X2` the result of `self` minus the argument.

`matrix2X2 operator*(double)`
Return a new `matrix2X2` with each element of `self` multiplied by the argument.

`matrix2X2 operator*(matrix2X2&)`
Return a new `matrix2X2` equal to the matrix product of `self` and the argument.

`matrix operator*(matrix&)`
Return a new `matrix` equal to the product of `self` and the argument.

`matrix2X2 operator!()`
Return a new `matrix2X2` equal to the transpose of `self`.

`double operator--()`
Return the determinant of `self`.
`matrix2X2 operator~()`
Return a new `matrix2X2` equal to the inverse of `self`.

Commentary

Class Xform_matrix:public matrix4X4

An Xform_matrix is a 4x4 matrix with additional operations that characterize its role as a 3-D graphics transformation matrix.

The Xform_type argument values may be chosen from {TRANSLATE, ROTATE, SCALE} and axis argument values are from {X, Y, Z}.
The hand arguments are from {L, R}.

Constructors

Xform_matrix()
: Create an identity transformation.

Xform_matrix(Xform_matrix& Xm)
: Create a duplicate of the argument.

Xform_matrix(matrix4X4& m4)
: Convert a general 4x4 matrix to an Xform_matrix.

Xform_matrix(Xform_type, double x, double y, double z=0.0)
: Create an Xform_matrix with the indicated 3-axis transformation.

Xform_matrix(Xform_type, axis, double)
: Create an Xform_matrix with the indicated 1-axis transformation.

Xform_matrix(gpoint&)
: Create a translation matrix from the given gpoint.

Xform_matrix(hand, gvector d, axis a1, gvector u, axis a2)
: Create an Xform_matrix with the indicated handedness, with direction vector d in column a1, and with up vector u, after adjustment, going into column a2. The remaining column is determined by the handedness and by cross products.

Member functions

Xform_matrix& identity()
: Make self an identity transformation.

Xform_matrix& zero()
: Make self a zero matrix.

Xform_matrix& inverse()
: Set self equal to its inverse.

Xform_matrix& transform(Xform_type, axis, double)
: Transform self by the indicated 1-axis transformation.

Xform_matrix& transform(Xform_type, double, double, double)
: Transform self by the indicated 3-axis transformation.

Xform_matrix& translate(double, double, double)
: Translate self as indicated on all 3 axes.

Xform_matrix& translate(axis, double)
: Translate self as indicated on one axis.

Xform_matrix& translate(gvector&)
: Translate self by the gvector.

Xform_matrix& scale(double, double, double)

Scale `self` as indicated on all 3 axes.

`Xform_matrix& scale(axis, double)`
Scale `self` as indicated on one axis.

`Xform_matrix& rotate(axis, double)`
Rotate `self` about one axis the given number of radians.

`Xform_matrix& shear(double, double, double)`
Shear `self` as indicated.

`Xform_matrix& perspective(double,double,double)`
Apply perspective transformation to `self` as indicated.

`Xform_matrix& ortho(hand, gvector, axis, gvector, axis)`
Apply the indicated orthogonal transformation to `self`.

`Xform_matrix& operator=(Xform_matrix&)`
Make `self` a duplicate of the argument.

`Xform_matrix& operator=(matrix4X4&)`
Make `self` a duplicate of the argument.

`Xform_matrix operator*(Xform_matrix&)`
Return a new `Xform_matrix` with the product of `self` and the argument.

`Xform_matrix& operator*=(Xform_matrix&)`
Multiply `self` by the argument.

`matrix operator*(matrix&)`
Return a new `matrix` holding the product of `self` and the argument.

`Xform_matrix operator~()`
Return a new `Xform_matrix` holding the inverse of `self`.

Commentary

Class Polyline:public matrix

A Polyline is a sequence of gpoints stored as a matrix.

Constructors

Polyline(Polyline& p)
: Construct a duplicate of the argument.

Polyline(int)
: Construct a Polyline with room for the indicated number of gpoints.

Member functions

int size()
: Return the number of gpointscurrently in self.

int maxsize()
: Return the maximum allocated size of self.

Polyline& reset()
: Set the internal index in self to zero.

gpoint get_next()
: Return the next gpoint in self and increment the index.

gpoint get(int)
: Return the indicated gpoint from self.

Polyline& put_next(gpoint)
: Put the given gpoint into self and increment the index.

Polyline& put(int,gpoint)
: Put the given gpoint into self at the indicated location.

Polyline* process(Xform_matrix)
: Create a new Polyline in which each gpoint of self has been transformed by the given Xform_matrix. Return a pointer to that new Polyline.

Polyline& homogenize()
: Divide through each gpoint in this Polyline by its w coordinate.

Polyline& done()
: This message indicates that no more additions will be made to self, so the data array can be copied into a new array whose size just fits the number of gpoints currently in self.

Polyline& operator=(Polyline&)
: Make self a copy of the given Polyline.

Commentary